DEPARTMENT OF MATHEMATICS AND COMPUTING

V-M.Tech. (M&C)

Monsoon Semester 2022-2023

GPU Computing Lab (MCC302)

LAB-7

Sum Reduction Algorithm

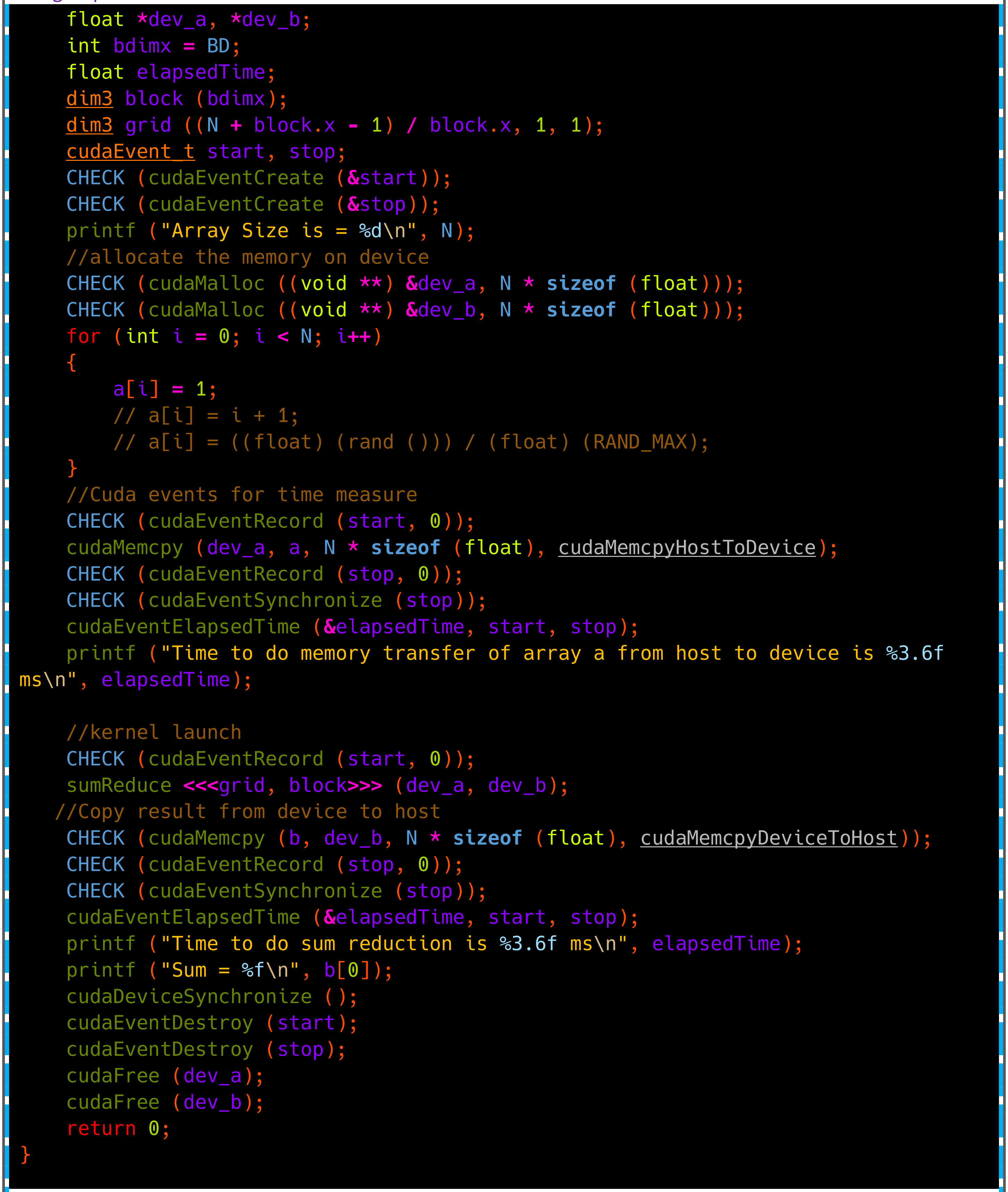
NAME: **ROMEO SARKAR**
ADMISSION NO.: **20JE0814**
DATE OF EXPERIMENT: **12-10-2022**

Experiment 2.1: Sum reduction

Objectives: Write a CUDA program for sum reduction.

CUDA Sample Program:

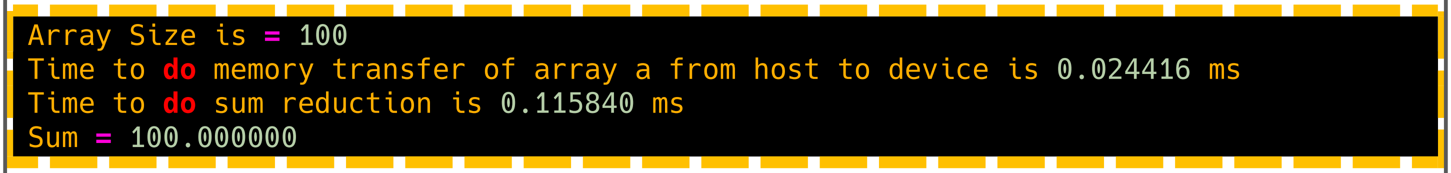
```
#include <cuda_runtime.h>
#include <stdio.h>
#define N 100
#define BD 256

#define CHECK(call) \
{\
    const cudaError_t error = call;\
    if (error != cudaSuccess)\
    {\
        fprintf (stderr, "error: %s: %d,", __FILE__, __LINE__);\
        fprintf (stderr, "code:%d, reason:%s\n", error, cudaGetErrorString
(error));\
        exit (1);\
    }\
}

__global__ void sumReduce (float *dev_a, float *dev_b)
{
    __shared__ float partialSum[BD];
    partialSum[threadIdx.x] = dev_a[blockIdx.x * blockDim.x + threadIdx.x];
    unsigned int t = threadIdx.x;
    for (unsigned int stride = 1; stride < blockDim.x; stride *= 2)
    {
        __syncthreads ();
        if ((t % (2 * stride)) == 0)
        {
            partialSum[t] += partialSum[t + stride];
        }
    }
    if (0 == threadIdx.x)
    {
        dev_b[blockIdx.x] = partialSum[0];
    }
    return;
}

int main (int argc, char **argv)
{
    float a[N], b[N];
```



```
    float *dev_a, *dev_b;
    int bdimx = BD;
    float elapsedTime;
    dim3 block (bdimx);
    dim3 grid ((N + block.x - 1) / block.x, 1, 1);
    cudaEvent_t start, stop;
    CHECK (cudaEventCreate (&start));
    CHECK (cudaEventCreate (&stop));
    printf ("Array Size is = %d\n", N);
    //allocate the memory on device
    CHECK (cudaMalloc ((void **) &dev_a, N * sizeof (float)));
    CHECK (cudaMalloc ((void **) &dev_b, N * sizeof (float)));
    for (int i = 0; i < N; i++)
    {
        a[i] = 1;
        // a[i] = i + 1;
        // a[i] = ((float) (rand ())) / (float) (RAND_MAX);
    }
    //Cuda events for time measure
    CHECK (cudaEventRecord (start, 0));
    cudaMemcpy (dev_a, a, N * sizeof (float), cudaMemcpyHostToDevice);
    CHECK (cudaEventRecord (stop, 0));
    CHECK (cudaEventSynchronize (stop));
    cudaEventElapsedTime (&elapsedTime, start, stop);
    printf ("Time to do memory transfer of array a from host to device is %3.6f
ms\n", elapsedTime);

    //kernel launch
    CHECK (cudaEventRecord (start, 0));
    sumReduce <<<grid, block>>> (dev_a, dev_b);
   //Copy result from device to host
    CHECK (cudaMemcpy (b, dev_b, N * sizeof (float), cudaMemcpyDeviceToHost));
    CHECK (cudaEventRecord (stop, 0));
    CHECK (cudaEventSynchronize (stop));
    cudaEventElapsedTime (&elapsedTime, start, stop);
    printf ("Time to do sum reduction is %3.6f ms\n", elapsedTime);
    printf ("Sum = %f\n", b[0]);
    cudaDeviceSynchronize ();
    cudaEventDestroy (start);
    cudaEventDestroy (stop);
    cudaFree (dev_a);
    cudaFree (dev_b);
    return 0;
}
```

SumReduction.cu

Output:

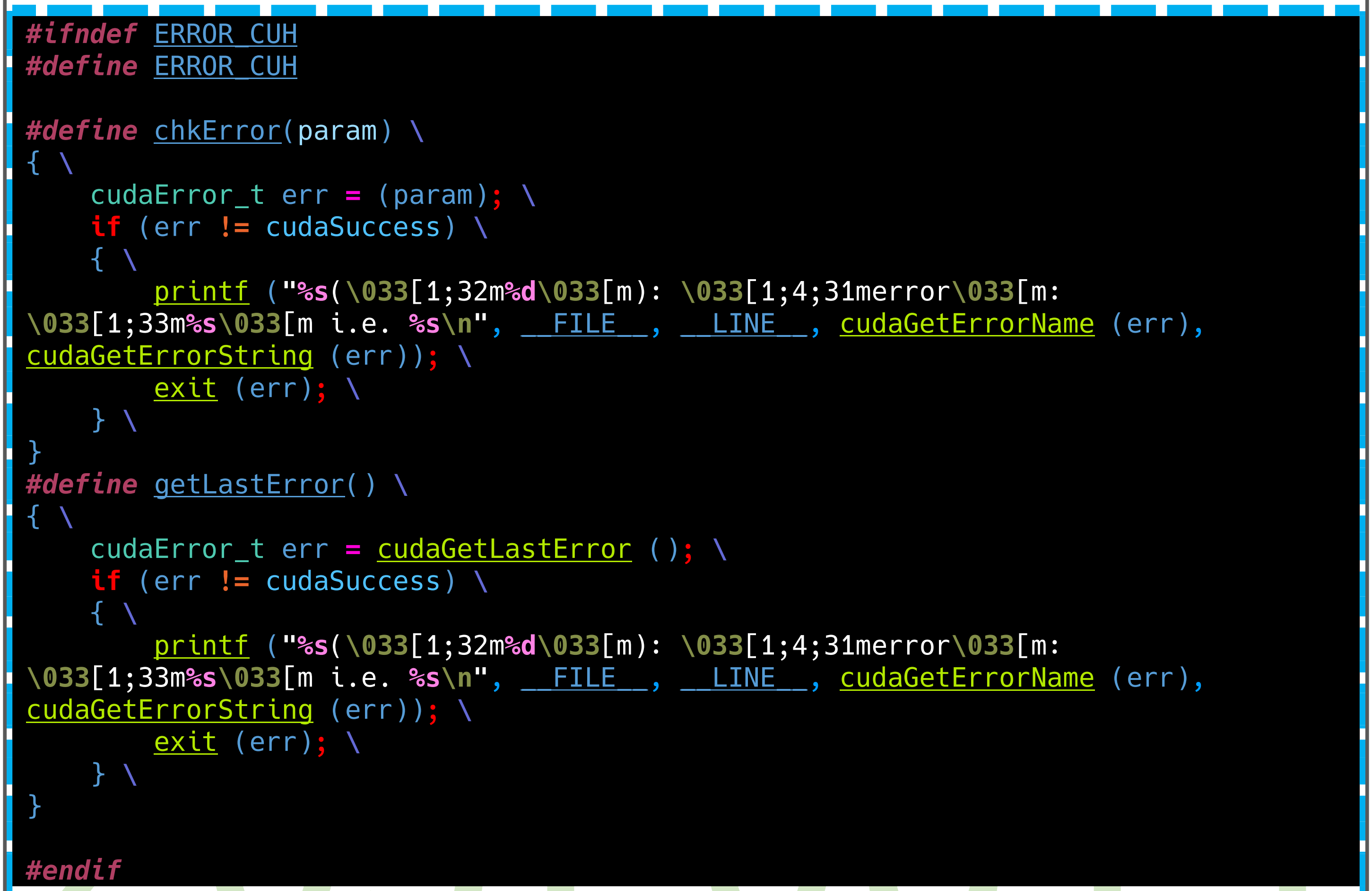


```
Array Size is = 100
Time to do memory transfer of array a from host to device is 0.024416 ms
Time to do sum reduction is 0.115840 ms
Sum = 100.000000
```

Lab Exercise 2.2: Write a CUDA program to demonstrate the followings:

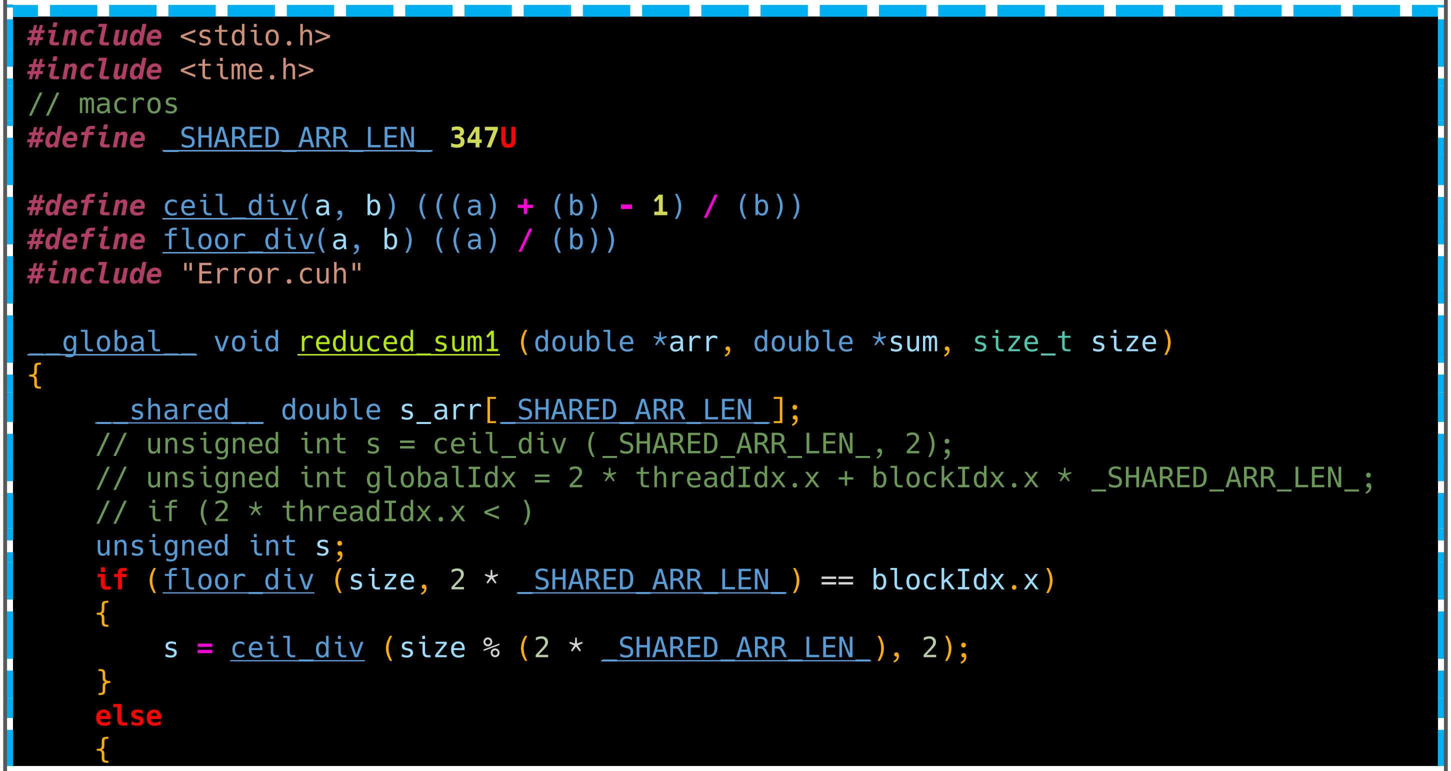
1. Write a header file for declaring Error function
2. Write device functions to do the sum reduction with less warp divergence
3. Print the execution time of the kernel and compare with classical sum reduction as given in 2.1
4. Print the result

CODE:

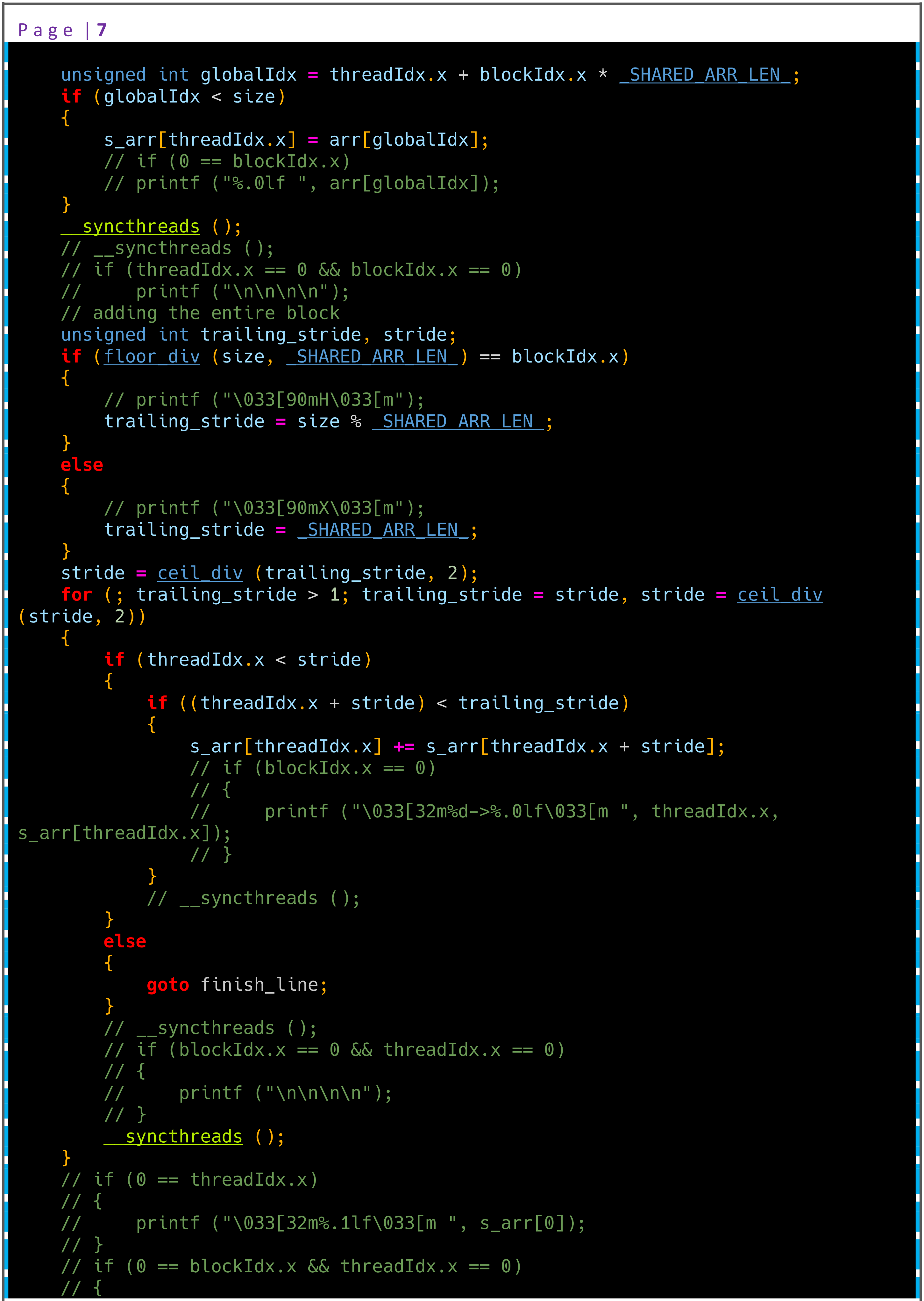


```
#ifndef ERROR_CUH
#define ERROR_CUH

#define chkError(param) \
{ \
    cudaError_t err = (param); \
    if (err != cudaSuccess) \
    { \
        printf ("%s(\033[1;32m%d\033[m): \033[1;4;31merror\033[m:
\033[1;33m%s\033[m i.e. %s\n", __FILE__, __LINE__, cudaGetErrorName (err),
cudaGetErrorString (err)); \
        exit (err); \
    } \
}
#define getLastError() \
{ \
    cudaError_t err = cudaGetLastError (); \
    if (err != cudaSuccess) \
    { \
        printf ("%s(\033[1;32m%d\033[m): \033[1;4;31merror\033[m:
\033[1;33m%s\033[m i.e. %s\n", __FILE__, __LINE__, cudaGetErrorName (err),
cudaGetErrorString (err)); \
        exit (err); \
    } \
}

#endif
```

Error.cuh



```
#include <stdio.h>
#include <time.h>
// macros
#define _SHARED_ARR_LEN_ 347U

#define ceil_div(a, b) (((a) + (b) - 1) / (b))
#define floor_div(a, b) ((a) / (b))
#include "Error.cuh"

__global__ void reduced_sum1 (double *arr, double *sum, size_t size)
{
    __shared__ double s_arr[_SHARED_ARR_LEN_];
    // unsigned int s = ceil_div (_SHARED_ARR_LEN_, 2);
    // unsigned int globalIdx = 2 * threadIdx.x + blockIdx.x * _SHARED_ARR_LEN_;
    // if (2 * threadIdx.x < )
    unsigned int s;
    if (floor_div (size, 2 * _SHARED_ARR_LEN_) == blockIdx.x)
    {
        s = ceil_div (size % (2 * _SHARED_ARR_LEN_), 2);
    }
    else
    {
        s = _SHARED_ARR_LEN_;
    }
    if ((threadIdx.x < s) && ((2 * (threadIdx.x + blockIdx.x * _SHARED_ARR_LEN_))
< size))
    {
        s_arr[threadIdx.x] = arr[2 * (threadIdx.x + blockIdx.x *
_SHARED_ARR_LEN_)];
    }
    // else
    // {
    //     goto finish_line;
    // }
    __syncthreads ();
    if ((threadIdx.x < s) && ((2 * (threadIdx.x + blockIdx.x * _SHARED_ARR_LEN_) +
1) < size))
    {
        s_arr[threadIdx.x] += arr[2 * (threadIdx.x + blockIdx.x *
_SHARED_ARR_LEN_) + 1];
    }
    // if (0 == blockIdx.x)
    // {
    //     // for (unsigned int i = 0; i < _SHARED_ARR_LEN_; i++)
    //     // {
    //     printf ("%.0lf ", s_arr[threadIdx.x]);
    //     // }
    //     __syncthreads ();
    //     if (0 == threadIdx.x)
    //         printf ("\n");
    // }
    __syncthreads ();

    // unsigned int
    // now, find the sum of the entire block:
    for (unsigned int stride = 1; stride < s; stride <<= 1)
    {
        if ((threadIdx.x % (stride << 1)) == 0)
        {
            if ((threadIdx.x + stride) < s)
            {
                s_arr[threadIdx.x] += s_arr[threadIdx.x + stride];
            }
        }
        // else
        // {
        //     goto finish_line;
        // }
        __syncthreads ();
    }
    sum[blockIdx.x] = s_arr[0];
    // finish_line:
    // printf ("<<%u;%u>>\n", blockIdx.x, threadIdx.x);
    // blockIdx.x
    return;
}
__global__ void reduced_sum2 (double *arr, double *sum, unsigned int size)
{
    __shared__ double s_arr[_SHARED_ARR_LEN_];
    // #define s_arr arr
```

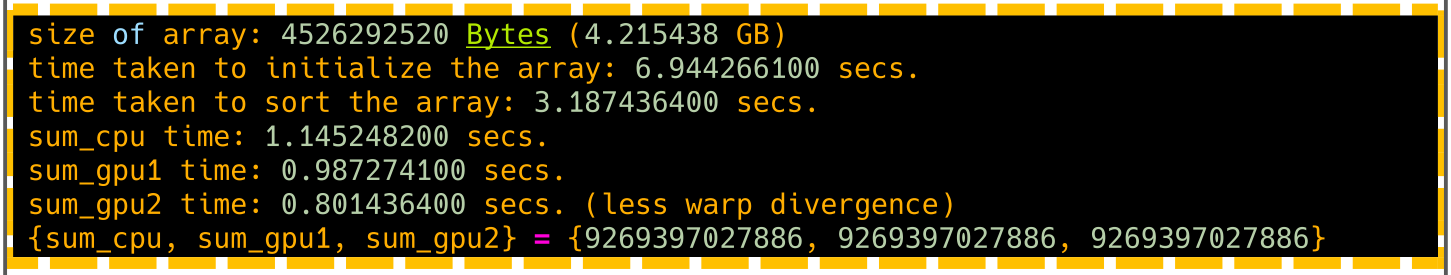


```
    unsigned int globalIdx = threadIdx.x + blockIdx.x * _SHARED_ARR_LEN_;
    if (globalIdx < size)
    {
        s_arr[threadIdx.x] = arr[globalIdx];
        // if (0 == blockIdx.x)
        // printf ("%.0lf ", arr[globalIdx]);
    }
    __syncthreads ();
    // __syncthreads ();
    // if (threadIdx.x == 0 && blockIdx.x == 0)
    //     printf ("\n\n\n\n");
    // adding the entire block
    unsigned int trailing_stride, stride;
    if (floor_div (size, _SHARED_ARR_LEN_) == blockIdx.x)
    {
        // printf ("\033[90mH\033[m");
        trailing_stride = size % _SHARED_ARR_LEN_;
    }
    else
    {
        // printf ("\033[90mX\033[m");
        trailing_stride = _SHARED_ARR_LEN_;
    }
    stride = ceil_div (trailing_stride, 2);
    for (; trailing_stride > 1; trailing_stride = stride, stride = ceil_div
(stride, 2))
    {
        if (threadIdx.x < stride)
        {
            if ((threadIdx.x + stride) < trailing_stride)
            {
                s_arr[threadIdx.x] += s_arr[threadIdx.x + stride];
                // if (blockIdx.x == 0)
                // {
                //     printf ("\033[32m%d->%.0lf\033[m ", threadIdx.x,
s_arr[threadIdx.x]);
                // }
            }
            // __syncthreads ();
        }
        else
        {
            goto finish_line;
        }
        // __syncthreads ();
        // if (blockIdx.x == 0 && threadIdx.x == 0)
        // {
        //     printf ("\n\n\n\n");
        // }
        __syncthreads ();
    }
    // if (0 == threadIdx.x)
    // {
    //     printf ("\033[32m%.1lf\033[m ", s_arr[0]);
    // }
    // if (0 == blockIdx.x && threadIdx.x == 0)
    // {
```

```
    //     printf ("\033[31m%lf\033[m\n", s_arr[0]);
    // }
    if (0 == threadIdx.x)
    {
        sum[blockIdx.x] = s_arr[0];
    }
    // sum[blockIdx.x] = arr[blockIdx.x * _SHARED_ARR_LEN_];
    finish_line:
    return;
}
double calculate_sum_cpu (double *arr, size_t size)
{
    double s = 0;
    for (size_t i = 0; i < size; i++)
    {
        s += arr[i];
    }
    return s;
}
double calculate_sum_cpu (double *arr, size_t startIdx, size_t endIdx)
{
    double s = 0;
    for (int i = startIdx; i < endIdx; i++)
    {
        s += arr[i];
    }
    return s;
}
void initialize_array (double *arr, size_t size)
{
    struct timespec start, stop;
    timespec_get (&start, TIME_UTC);
    for (size_t i = 0; i < size; i++)
    {
        // arr[i] = ((double) rand ()) * ((double) (rand ()));
        arr[i] = (double) rand ();
        // arr[i] = i + 1;
        // arr[i] = 1;
    }
    timespec_get (&stop, TIME_UTC);
    printf ("time taken to initialize the array: %.9lf secs.\n", ((double)
(stop.tv_nsec - start.tv_nsec) * 1e-9 + ((double) (stop.tv_sec - start.tv_sec))));
    return;
}
int cmp (const void *a, const void *b)
{
    const double *x = (const double *) (a), *y = (const double *) (b);
    if (x < y)
    {
        return 0; // i.e. don't swap
    }
    else
    {
        return 1; // i.e. swap
    }
}
void sort_array (double *arr, size_t size)
{
    struct timespec start, stop;
    timespec_get (&start, TIME_UTC);
    qsort (arr, size, sizeof (double), cmp);
    timespec_get (&stop, TIME_UTC);
    printf ("time taken to sort the array: %.9lf secs.\n", ((double) (stop.tv_nsec
- start.tv_nsec)) * 1e-9 + ((double) (stop.tv_sec - start.tv_sec)));
    return;
}
void print_array (double *arr, size_t size)
{
    for (size_t i = 0; i < size; i++)
    {
        printf ("%.0f ", arr[i]);
    }
    printf ("\n");
    return;
}
double calculate_sum_gpu1 (double *arr, size_t size)
{
    // double *dev_arr;
    // cudaMalloc (&dev_arr, size);

    // cudaMemcpy (dev_arr, arr, size * sizeof (double), cudaMemcpyHostToDevice);
    // array will be divided into smaller array of size _SHARED_ARR_LEN_
    double sum;
    size_t temp_arr_size = size, temp_sum_arr_size = ceil_div (temp_arr_size, 2 *
_SHARED_ARR_LEN_);
    double *dev_temp_arr = NULL, *dev_temp_sum_arr = NULL;
    cudaMalloc (&dev_temp_arr, sizeof (double) * temp_arr_size);
    cudaMemcpy (dev_temp_arr, arr, temp_arr_size * sizeof (double),
cudaMemcpyHostToDevice);
    for (; temp_arr_size > 1; temp_arr_size = temp_sum_arr_size, temp_sum_arr_size
= ceil_div (temp_arr_size, 2 * _SHARED_ARR_LEN_))
    {
        // temp_size = ceildiv (temp_size, _SHARED_ARR_LEN_);
        // temp_arr = (double *) (malloc (sizeof (double) * temp_arr_size));
        // printf ("launch param: <<< %zu, %u >>>\n", temp_sum_arr_size,
_SHARED_ARR_LEN_);
        // printf ("launch param1: %u\n", ceil_div (12, 5));
        chkError (cudaMalloc (&dev_temp_sum_arr, sizeof (double) *
temp_sum_arr_size))
        reduced_sum1 <<< temp_sum_arr_size, _SHARED_ARR_LEN_ >>> (dev_temp_arr,
dev_temp_sum_arr, temp_arr_size);
        getLastError ();
        cudaDeviceSynchronize ();
        // printf ("launch param2: %zu, %zu\n", temp_arr_size, temp_sum_arr_size);
        // printf ("\n");
        // comment:
        // double *p = (double *) malloc (temp_sum_arr_size * sizeof (double));
        // cudaMemcpy (p, dev_temp_sum_arr, sizeof (double) * temp_sum_arr_size,
cudaMemcpyDeviceToHost);
        // double t;
        // for (int i = 0; i < temp_sum_arr_size; i++)
        // {
        //     printf ("\033[31m%.1lf\033[m ", p[i]);
        // }
        // printf ("\n");
        // for (int i = 0; i < temp_sum_arr_size; i++)
        // {
        //     if (p[i] != (t = calculate_sum_cpu (arr, i * 2 * _SHARED_ARR_LEN_,
(i + 1) * 2 * _SHARED_ARR_LEN_)))
        //     {
        //         printf ("error: %d; %.1lf instead of %.1lf\n", i, p[i], t);
        //         exit (0);
        //     }
        // }
        // exit (0);
        // comment:

        cudaFree (dev_temp_arr);
        dev_temp_arr = dev_temp_sum_arr;
        dev_temp_sum_arr = NULL;
        // return 0;

    }
    cudaMemcpy (&sum, dev_temp_arr, sizeof (double), cudaMemcpyDeviceToHost);
    cudaFree (dev_temp_arr);
    return sum;
}
double calculate_sum_gpu2 (double *arr, size_t size)
{
    // double *dev_arr;
    // cudaMalloc (&dev_arr, size);

    // cudaMemcpy (dev_arr, arr, size * sizeof (double), cudaMemcpyHostToDevice);
    // array will be divided into smaller array of size _SHARED_ARR_LEN_
    double sum;
    size_t temp_arr_size = size, temp_sum_arr_size = ceil_div (size,
_SHARED_ARR_LEN_);
    double *dev_temp_arr = NULL, *dev_temp_sum_arr = NULL;
    cudaMalloc (&dev_temp_arr, sizeof (double) * temp_arr_size);
    cudaMemcpy (dev_temp_arr, arr, temp_arr_size * sizeof (double),
cudaMemcpyHostToDevice);
    for (; temp_arr_size > 1; temp_arr_size = temp_sum_arr_size, temp_sum_arr_size
= ceil_div (temp_sum_arr_size, _SHARED_ARR_LEN_))
    {
        // temp_size = ceildiv (temp_size, _SHARED_ARR_LEN_);
        // temp_arr = (double *) (malloc (sizeof (double) * temp_arr_size));
        cudaMalloc (&dev_temp_sum_arr, sizeof (double) * temp_sum_arr_size);
        reduced_sum2 <<< temp_sum_arr_size, _SHARED_ARR_LEN_ >>> (dev_temp_arr,
dev_temp_sum_arr, temp_arr_size);
        getLastError ();
        cudaDeviceSynchronize ();
        // printf ("\n");
        // comment:
        // double *p = (double *) malloc (temp_sum_arr_size * sizeof (double));
        // cudaMemcpy (p, dev_temp_sum_arr, sizeof (double) * temp_sum_arr_size,
cudaMemcpyDeviceToHost);
        // double t;
        // for (int i = 0; i < temp_sum_arr_size && i < 1; i++)
        // {
        //     if (p[i] != (t = calculate_sum_cpu (arr, i * _SHARED_ARR_LEN_, (i +
1) * _SHARED_ARR_LEN_)))
        //     {
        //         printf ("error: %d; %.1lf instead of %.1lf\n", i, p[i], t);
        //     }
        // }
        // comment:
        /*
        f
        sf
        sd

        */

        cudaFree (dev_temp_arr);
        dev_temp_arr = dev_temp_sum_arr;
        dev_temp_sum_arr = NULL;
        // return 0;

    }
    cudaMemcpy (&sum, dev_temp_arr, sizeof (double), cudaMemcpyDeviceToHost);
    cudaFree (dev_temp_arr);
    return sum;
}
int sum (double *arr, size_t size)
{
    struct timespec start, stop;
    // timespec_get (&start, TIME_UTC);
    // clock_t st = clock ();
    sort_array (arr, size);
    // printf ("time: %.3lf secs.\n", ((double) (clock () - st)) /
CLOCKS_PER_SEC);
    // timespec_get (&stop, TIME_UTC);
    // printf ("time taken to sort the array: %.9lf secs.\n", ((double)
(stop.tv_nsec - start.tv_nsec)) * 1e-9 + ((double) (stop.tv_sec - start.tv_sec)));
    /* = = = = = = = = = = = = = = */
    timespec_get (&start, TIME_UTC);
    double sum_cpu = calculate_sum_cpu (arr, size);
    timespec_get (&stop, TIME_UTC);
    printf ("sum_cpu time: %.9lf secs.\n", ((double) (stop.tv_nsec -
start.tv_nsec) * 1e-9 + ((double) (stop.tv_sec - start.tv_sec))));
    /*= = = = = = = = = = = = = = */
    timespec_get (&start, TIME_UTC);
    double sum_gpu1 = calculate_sum_gpu1 (arr, size);
    timespec_get (&stop, TIME_UTC);
    printf ("sum_gpu1 time: %.9lf secs.\n", ((double) (stop.tv_nsec -
start.tv_nsec) * 1e-9 + ((double) (stop.tv_sec - start.tv_sec))));
    /*= = = = = = = = = = = = = = */
    timespec_get (&start, TIME_UTC);
    double sum_gpu2 = calculate_sum_gpu2 (arr, size);
    timespec_get (&stop, TIME_UTC);
    printf ("sum_gpu2 time: %.9lf secs. \033[90m(less warp divergence)\033[m\n",
((double) (stop.tv_nsec - start.tv_nsec) * 1e-9 + ((double) (stop.tv_sec -
start.tv_sec))));
    /*= = = = = = = = = = = = = = */
    printf ("{sum_cpu, sum_gpu1, sum_gpu2} = {%.0lf, %.0lf, %.0lf}\n", sum_cpu,
sum_gpu1, sum_gpu2);
    if (sum_cpu != sum_gpu1)
    {
        printf ("\033[1;31merror\033[m: (sum_cpu != sum_gpu1)\n");
        return 1;
    }
    else
    {
        if (sum_cpu != sum_gpu2)
        {
            printf ("\033[1;31merror\033[m: (sum_cpu != sum_gpu2)\n");
            return 1;
        }
        else
        {
            return 0;
        }
    }
    // const int i = 0;
    // if (sum_cpu != sum_gpu2)
    // {
    //     printf ("\033[1;31merror\033[m: (sum_cpu != sum_gpu)\n");
    //     return 1;
    // }
    // else
    // {
    //     return 0;
    // }
    return 0;
}
double *allocate_array (size_t size)
{
    printf ("size of array: %zd Bytes (%.6lf GB)\n", sizeof (double) * size,
((double) (sizeof (double) * size)) / (1024.0 * 1024.0 * 1024.0));
    double *arr = (double *) (malloc (sizeof (double) * size));
    return arr;
}
int main ()
{
    srand (time (NULL));
    size_t size = 565786565; // array size;

    double *arr = allocate_array (size);
    initialize_array (arr, size);
    sum (arr, size);
    return 0;
}
```

SumReduction.cu

Output:



```
size of array: 4526292520 Bytes (4.215438 GB)
time taken to initialize the array: 6.944266100 secs.
time taken to sort the array: 3.187436400 secs.
sum_cpu time: 1.145248200 secs.
sum_gpu1 time: 0.987274100 secs.
sum_gpu2 time: 0.801436400 secs. (less warp divergence)
{sum_cpu, sum_gpu1, sum_gpu2} = {9269397027886, 9269397027886, 9269397027886}
```